Rapport de Projet
Simulateur de Bourse

Sommaire

1 Introduction **2**

2 Fonctionnalités principales **2**

3 Architecture **4**
3.1 Interface graphique 4
3.1.1 Diagramme de classes 4
3.1.2 Explication des classes 4
3.2 Parties Utilisateur et gestion financière 7
3.2.1 Diagramme de classes 7
3.2.2 Explication des classes 7

4 Dynamique du système **11**
4.1 Collaboration entre objets 11
4.2 Circulation des données dans le système 11
4.3 Dépendances d'éxecutions 11

5 Principaux choix de conception et réalisation **12**
5.1 Importation et stockage des données 12
5.2 Mise à jour dynamique des données et interaction utilisateur 12

6 Organisation du développement **13**

Réalisé par :
Ralph Khairallah

Date : June 5, 2025

1 Introduction

Dans ce projet personnel, j'ai choisi de développer une application intitulée **Simulateur de Bourse**. L'objectif principal est de concevoir un outil pédagogique permettant à tout type d'utilisateur, débutant ou initié, d'apprendre à investir virtuellement en bourse.

À travers une interface simple mais complète, l'utilisateur peut acheter et vendre des actions fictives, suivre l'évolution de son portefeuille, et observer en temps réel les effets de ses décisions. Ce simulateur a pour ambition de rendre plus accessibles les notions clés du monde de l'investissement : la gestion des risques, le suivi de performance, et la stratégie de placement dans différentes situations du marché boursier, comme un krach financier, une période de forte inflation, ou des événements exceptionnels comme la crise du COVID-19.

Ce rapport présente les différentes étapes de conception et de développement de l'application. Il détaille l'architecture mise en place, les choix techniques adoptés au fil du temps, les principales fonctionnalités implémentées, ainsi que les difficultés rencontrées et les solutions proposées.

2 Fonctionnalités principales

Les principales user-stories développées dans cette application sont les suivantes :

Importer les données

L'objectif de cette user-story est de pouvoir importer efficacement des données de produits financiers sur de longues périodes de temps. Par exemple, pour cette application, il est essentiel d'avoir accès aux prix d'ouverture et de fermeture (prix au début et à la fin d'une journée) d'une action précise sur une période donnée.

Cette fonctionnalité a été réalisée en priorité au début du développement et a été complétée.

Stocker les données

Une fois que les données ont été récupérées, il faut pouvoir les stocker de manière efficace afin d'y avoir facilement accès par la suite et de pouvoir les réutiliser.

Cette fonctionnalité a été mise en place progressivement. Certains points mériteraient encore d'être optimisés, mais le stockage est fonctionnel et permet d'exploiter les données dans l'application.

Afficher le cours d'un produit financier

Pour acheter un produit financier, l'utilisateur doit pouvoir suivre l'évolution de son prix dans le temps. L'objectif ici est de proposer un affichage dynamique des courbes de prix avec différents boutons permettant d'afficher ou de retirer certains indicateurs financiers.

L'affichage principal a été mis en place dès les premières phases du développement, puis enrichi avec des boutons d'interaction dans une phase ultérieure.

Acheter et vendre un produit financier

L'utilisateur doit pouvoir acheter ou vendre un produit financier de son choix. Le solde du portefeuille est automatiquement ajusté en fonction du prix d'achat ou de vente, et les actifs sont ajoutés ou retirés de la liste de possessions.

Cette fonctionnalité a été entièrement développée par la suite.

Accéder à n'importe quel produit financier

L'application intègre une barre de recherche permettant d'accéder à n'importe quel produit financier. Cela permet à l'utilisateur une grande liberté de navigation, sans être limité à une liste prédéfinie.

Cette fonctionnalité est opérationnelle, bien que certains aspects puissent encore être améliorés.

Créer un compte et se reconnecter à une partie en cours

L'utilisateur peut créer un compte afin de sauvegarder sa progression et y accéder ultérieurement. Cela nécessite de stocker des données liées à l'utilisateur, comme son nom d'utilisateur et son mot de passe.

Les bases de cette fonctionnalité ont été posées : les interfaces d'inscription et de connexion sont fonctionnelles, de même que le stockage des identifiants. En revanche, le système complet d'authentification reste à fiabiliser.

Avancer dans le temps

Plutôt que de suivre le marché en temps réel, l'application propose une simulation accélérée. Lorsqu'un utilisateur sélectionne une action, celle-ci s'affiche sur une échelle temporelle prédéfinie (ex. janvier à mars 2025). Il est alors possible d'avancer dans le temps pour observer immédiatement l'évolution de cette action.

Ce système est en place, mais ne permet pas encore de choisir librement la date de départ : les données s'affichent toujours à partir du début de 2025.

Se placer au cours d'un évènement historique

Dans une optique pédagogique, l'application doit permettre à l'utilisateur de se positionner directement sur une période marquée par un évènement historique majeur, afin d'en observer les impacts sur les marchés.

Cette fonctionnalité n'a pas encore été développée.

3 Architecture

3.1 Interface graphique

3.1.1 Diagramme de classes

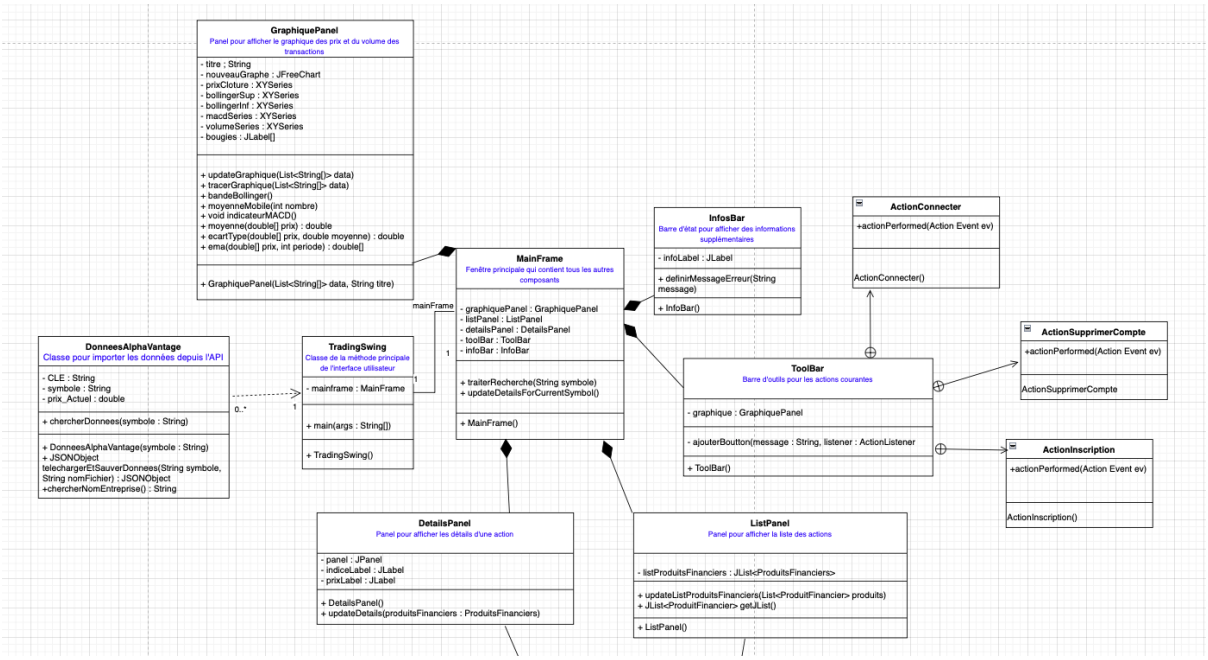

Figure 1: Partie du diagramme UML pour l'interface graphique

3.1.2 Explication des classes

➢ **GraphiquePanel** est un `JPanel` qui intègre deux graphiques générés avec JFreeChart :

- le graphique principal affiche l'évolution du prix de clôture dans le temps ;
- un graphique secondaire affiche les volumes.

Lorsqu'on appelle `setData(List<String[]> data)`, le graphique se met à jour avec les nouvelles données de l'action. La fenêtre de visualisation peut être agrandie avec le bouton "Avancer".

La classe calcule aussi le **prix visible sur le graphique** (via `prixGraphe`) en fonction de la date affichée, ce qui est essentiel si on veut acheter une action au prix du graphique.

`GraphiquePanel` utilise un mécanisme d'`Observer` avec `PropertyChangeSupport` pour avertir les autres composants (comme `MainFrame`) lorsque le prix du graphique change.

➢ **DonneAlphaVantage** est une classe qui sert à interagir avec l'API web `Alpha Vantage`. Elle est responsable de :

- `chercherDonne()` : méthode qui récupère les données journalières (open, high, low, close, volume) via API ou depuis un fichier JSON local (cache) ;
- `chercherNomEntreprise()` : méthode qui interroge l'API OVERVIEW pour obtenir le nom complet de l'entreprise (à partir de son symbole).

Les données sont stockées dans un fichier local pour limiter les appels à l'API (limités par jour). À chaque recherche, on vérifie si le fichier local est encore valide (en date), sinon on télécharge de nouveau les données.

➢ **MainFrame** est la classe principale de l'interface graphique. Elle hérite de `JFrame` et regroupe tous les composants (toolbar, graphique, panneau de gauche, panneau de droite, barre d'information, etc.).

Elle orchestre les interactions utilisateur et met à jour les composants en conséquence. Ses responsabilités incluent :

- `traiterRecherche(String symbole)` : méthode déclenchée lorsqu'on recherche une action. Elle :
  - utilise `DonneAlphaVantage` pour charger les données ;
  - met à jour le graphique via `graphiquePanel.setData(...)` ;
  - extrait le prix du graphique (ajusté dans le temps) et met à jour `detailsPanel` ;
  - ajoute l'action à la liste de gauche si elle n'y figure pas encore.
- `updateDetailsForCurrentSymbol()` : écoute les changements du graphique et met à jour dynamiquement les détails de l'action.

➢ **TradingSwing** est la classe contenant la **méthode main**. Elle initialise l'interface utilisateur en créant une instance de `MainFrame`, le point d'entrée graphique de l'application. Elle utilise `SwingUtilities.invokeLater` pour respecter les bonnes pratiques de Swing (exécution du code sur le thread graphique).

➢ **ToolBar** est une barre d'outils graphique qui regroupe tous les boutons d'actions disponibles dans l'application (recherche, indicateurs, connexion, inscription, etc.). Elle hérite de `JToolBar` et s'affiche en haut de la fenêtre.

Elle contient notamment :

- un champ de texte `champRecherche` dans lequel l'utilisateur peut saisir un symbole d'action à rechercher ;
- un bouton `Rechercher` qui déclenche la méthode `traiterRecherche()` de la classe `MainFrame` pour afficher les données de l'action demandée ;
- des boutons pour afficher les indicateurs graphiques comme les bandes de Bollinger ou la moyenne mobile ;
- les boutons d'inscription et de connexion, avec leur propre interface.

Le champ de recherche est également déclenché lorsqu'on tape sur la touche "Entrée" grâce à un `ActionListener`. Les erreurs (symbole invalide) ou succès (affichage graphique) sont relayés via la `InfoBar`.

➢ **InfoBar** est une simple barre textuelle située en bas de la fenêtre. Elle hérite de `JLabel` et permet d'afficher soit :

- des messages informatifs (ex. : "Données pour MCD") via `mettreAJourInfo()`;
- des messages d'erreur avec style particulier (fond jaune, texte rouge, bordure) via `definirMessageErreur()`.

Elle sert de feedback utilisateur rapide à chaque interaction.

➢ **ListPanel** représente la colonne de gauche de l'interface utilisateur. Elle est organisée selon un `BorderLayout` et regroupe trois composants essentiels :

- une `JList<ProduitFinancier>` en haut, qui affiche les produits financiers que l'utilisateur a consultés via la recherche ;
- la vue du portefeuille utilisateur (`VuePortefeuille`) au centre ;
- le contrôleur de portefeuille (`ControlerPortefeuille`) en bas, pour acheter ou vendre les produits.

Cette structure reflète clairement l'architecture **MVC** :

- le modèle est représenté par le `Portefeuille` ;
- la vue est `VuePortefeuille` ;
- le contrôleur est `ControlerPortefeuille`.

La méthode `updateListProduitsFinanciers(...)` permet de recharger entièrement la liste d'actions visibles. Lorsqu'un produit est recherché et non encore listé, il est ajouté à ce composant.

➢ **DetailsPanel** est un panneau graphique situé à droite de l'interface utilisateur. Il fournit un **résumé visuel** des données principales liées à un produit financier sélectionné :

- **symbole** de l'action (ex. : "AAPL", "TSLA") ;
- **prix courant** du jour ;
- **prix affiché sur le graphique**, si celui-ci est ajusté dans le temps ;
- **variation en pourcentage** entre aujourd'hui et la veille.

Tous ces éléments sont mis à jour via la méthode `updateDetails()` lorsqu'on sélectionne un produit dans la liste ou lorsqu'on effectue une recherche.

`DetailsPanel` permet ainsi à l'utilisateur d'avoir une vue synthétique et immédiate de l'état actuel de l'action en question, sans devoir analyser le graphique.

3.2 Parties Utilisateur et gestion financière

3.2.1 Diagramme de classes

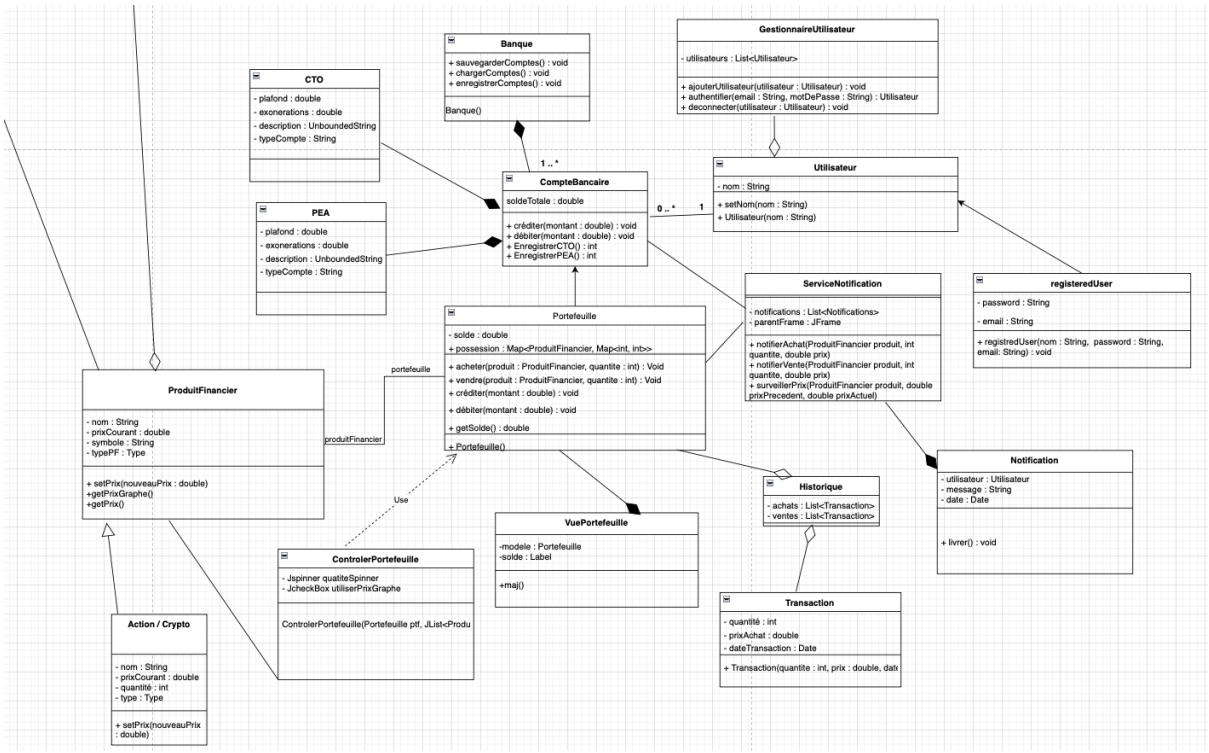

Figure 2: Partie du diagramme UML pour les parties utilisateur et gestion financière

3.2.2 Explication des classes

➢ **UserStorage** est une classe responsable de la **gestion en mémoire** des utilisateurs. Elle maintient deux listes : une liste générale `users` (tous les utilisateurs) et une liste `registredUsers` (utilisateurs enregistrés avec email/mot de passe). Elle fournit des méthodes pour :

- ajouter un utilisateur ;
- supprimer un utilisateur (par pseudo ou email) ;
- récupérer les utilisateurs stockés ;
- afficher les utilisateurs.

Cette classe est utile pour centraliser l'enregistrement et la gestion des comptes en local avant leur éventuelle persistance.

➢ **User** est la classe de base représentant un utilisateur de l'application. Il possède un `pseudo` et un `CompteBancaire`. C'est à travers ce compte qu'il interagit avec les portefeuilles. Il constitue l'identité minimale d'un utilisateur.

➢ **RegistredUser** hérite de `User` et ajoute des informations liées à l'authentification: `email` et `password`. Il est utilisé lors de la connexion et l'inscription. Cela permet de faire la distinction entre un utilisateur authentifié ou non.

- **Notification** représente un message destiné à l'utilisateur, daté et typé (achat, vente, alerte, etc.). Chaque notification contient un `message`, un `type`, et une `timestamp`. C'est une structure simple pour tracer les événements significatifs.

- **ServiceNotification** centralise la **gestion des notifications utilisateurs**. Il stocke une liste de notifications, et propose :

  - la création et l'affichage de messages pour chaque **achat ou vente** ;
  - l'envoi d'alertes en cas de **variation de prix importante** ;
  - l'affichage graphique des alertes via `JOptionPane`.

  Il fonctionne en étroite collaboration avec `Portefeuille` et la `MainFrame`.

- **CompteBancaire** modélise un compte utilisateur contenant plusieurs `Portefeuille`, `CTO`, ou `PEA`. Il permet de :

  - **créditer** ou **débiter** le solde global ;
  - **enregistrer des portefeuilles**, en attribuant automatiquement une clé ;
  - gérer l'état des avoirs de l'utilisateur.

  Il représente l'interface entre les produits financiers et les actions de l'utilisateur.

- **Transaction** décrit une opération d'achat ou de vente réalisée par l'utilisateur. Elle contient :

  - la `quantité` de titres échangés ;
  - le `prix d'achat unitaire` ;
  - la `date de la transaction`.

  Elle est utilisée par le `Portefeuille` pour suivre l'historique des achats.

- **GestionnaireUtilisateur** est une classe utilitaire responsable de la **persistance des utilisateurs**. Elle utilise la bibliothèque `Gson` pour sauvegarder et charger les utilisateurs dans un fichier JSON (`utilisateur.json`). Elle contient aussi :

  - une méthode pour connecter un utilisateur à partir d'un email et mot de passe ;
  - une méthode pour enregistrer un utilisateur après son inscription.

  Elle assure ainsi une passerelle entre le stockage local et l'état dynamique de l'application.

- **CTO** et **PEA** représentent deux types de comptes d'investissement, avec chacun ses plafonds fiscaux, ses exonérations et sa description réglementaire. Elles héritent toutes deux de `CompteBancaire`.

- **Banque** organise la création et l'ouverture de ces différents comptes.

- À l'intérieur d'un `CompteBancaire`, le **Portefeuille** prend en charge la détention et la transaction de produits financiers. Il conserve l'état actuel du portefeuille (la quantité de chaque titre) et fournit des méthodes pour acheter, vendre, créditer ou débiter.

➢ **Portefeuille** représente le cœur du modèle pour un utilisateur. Il gère à la fois :

- le **solde en liquidités** disponible pour effectuer des transactions ;
- la **possession actuelle** des titres financiers (quantité détenue, prix d'achat, etc.) ;
- les **transactions réalisées**.

Il utilise une map `Map<ProduitFinancier, List<Transaction>` pour suivre les quantités achetées et les prix associés. Chaque clé est un produit (action, crypto, etc.), et la valeur est la liste des `Transaction` effectuées sur ce produit.

Il propose plusieurs méthodes principales :

- `acheter(ProduitFinancier, int, double)` : ajoute une transaction d'achat après vérification du solde ;
- `vendre(ProduitFinancier, int)` : retire une quantité détenue si disponible, et crédite le solde ;
- `crediter()` et `debiter()` : manipulent directement le solde ;
- `getQuantite(...)` : retourne le nombre total d'unités possédées d'un produit ;
- `getPossessions()` : donne accès à la map des titres détenus.

La classe hérite de `Observable`, ce qui permet à la `VuePortefeuille` de s'abonner au modèle. Toute modification (achat, vente, crédit, débit) déclenche automatiquement une mise à jour de la vue graphique.

En cas d'achat ou de vente, `Portefeuille` envoie aussi une notification au système via `ServiceNotification`, notamment pour avertir des changements de prix ou enregistrer une alerte importante.

➢ **ProduitFinancier** est une classe `abstraite` qui représente tout actif pouvant être détenu dans un portefeuille : une action, une crypto-monnaie, un ETF, etc.

Elle définit les propriétés et comportements communs à tous les produits :

- `getNom()` et `getSymbole()` : fournissent respectivement le nom de l'actif et son code boursier ;
- `getPrixCourant()` : retourne le prix actuel de l'actif ;
- `getPrixGraphe()` : retourne le prix au moment affiché dans le graphique ;
- `setPrixCourant(...)` et `setPrixGraphe(...)` : permettent de mettre à jour dynamiquement les prix affichés.

Cette classe est conçue pour être étendue. Par exemple :

- `ActionGenerique` est une sous-classe concrète utilisée pour les actions recherchées dynamiquement par l'utilisateur ;
- `CryptoMonnaie` ou d'autres produits peuvent être implémentés ultérieurement avec la même structure.

Grâce à cette abstraction, l'application peut manipuler tous les types de titres de manière uniforme, que ce soit pour les afficher, les acheter, ou les analyser graphiquement.

➢ **VuePortefeuille**

La classe `VuePortefeuille` représente la **vue graphique du portefeuille de l'utilisateur**. Elle hérite de `JPanel` et affiche deux éléments principaux :

- un `JLabel` qui indique le **solde actuel** de l'utilisateur ;
- un `JTable` qui liste les produits financiers détenus (nom, quantité, prix d'achat).

`VuePortefeuille` observe le modèle `Portefeuille`, qui hérite de `Observable`. Lorsqu'une transaction est effectuée (achat ou vente), le modèle notifie automatiquement la vue, qui appelle la méthode `maj()` pour mettre à jour l'affichage.

`maj()` parcourt la map des possessions du portefeuille et recharge entièrement le tableau. Chaque ligne affiche :

- le nom du produit (`ProduitFinancier.getNom()`) ;
- la quantité possédée (extrait depuis les `Transaction`) ;
- le prix d'achat.

Ainsi, `VuePortefeuille` constitue une interface visuelle fidèle et synchronisée avec les données du modèle.

➢ **ControlerPortefeuille**

La classe `ControlerPortefeuille` est un composant graphique qui sert de **contrôleur d'interaction pour le portefeuille**. Elle permet à l'utilisateur d'effectuer des opérations d'achat et de vente sur les produits financiers sélectionnés dans l'interface.

Elle contient :

- un `JSpinner` pour choisir la quantité à acheter ou vendre ;
- une `JCheckBox` permettant de sélectionner l'utilisation du **prix du graphique** (plutôt que le prix courant) ;
- deux `JButton` : "Acheter" et "Vendre".

Lorsqu'un des boutons est cliqué :

1. le produit sélectionné dans la `JList<ProduitFinancier>` est récupéré ;
2. la quantité est lue dans le `JSpinner` ;
3. le prix utilisé dépend de l'état de la `JCheckBox` (`getPrixGraphe()` ou `getPrixCourant()`) ;
4. la méthode `ptf.acheter(...)` ou `ptf.vendre(...)` est appelée avec les bons paramètres ;
5. si la transaction est acceptée, la vue du portefeuille est automatiquement rafraîchie.

Des boîtes de dialogue sont affichées en cas d'erreur (produit non sélectionné, fonds insuffisants, ou quantité invalide).

`ControlerPortefeuille` et `VuePortefeuille` sont tous deux intégrés dans le `ListPanel`, assurant une séparation claire entre la logique métier (modèle), les interactions (contrôleur) et l'affichage (vue).

4 Dynamique du système

4.1 Collaboration entre objets

Lorsqu'un utilisateur souhaite acheter un produit financier, l'interaction débute par l'interface utilisateur, via **ControlerPortefeuille**, où l'utilisateur clique sur un bouton dédié. Cet événement est capté par le **MainFrame**, qui agit comme point d'entrée et de coordination de l'application.

Le **MainFrame** fait appel au portefeuille associé à l'utilisateur, représenté par la classe **Portefeuille**, pour initier l'opération d'achat. Ce dernier, à son tour, consulte le **CompteBancaire** pour vérifier que les fonds sont suffisants, puis procède au débit du montant requis.

Ensuite, le portefeuille met à jour la possession de produits financiers, et crée une instance de la classe **Transaction** pour archiver l'achat. Cette transaction est ensuite ajoutée à l'**Historique** de l'utilisateur. En parallèle, une notification est générée via le **ServiceNotification** et adressée à l'utilisateur pour l'informer de la réussite de l'opération.

Enfin, le **MainFrame**, à travers l'objet **InfoBar**, affiche un message de confirmation visuel.

4.2 Circulation des données dans le système

Les données circulent dans le simulateur de manière structurée. Les cours des produits financiers, par exemple, sont récupérés au démarrage et en appuyant sur le bouton "Avancer" de la **ToolBar**, par la classe **DonnéesAlphaVantage**, qui se connecte à l'API externe **AlphaVantage**.

Ces données sont ensuite stockées dans les objets **ProduitFinancier**, puis exploitées par les interfaces comme **GraphiquePanel** ou **ListPanel** pour les représenter graphiquement ou sous forme de liste.

Lorsqu'un utilisateur effectue un achat, les données saisies dans l'interface deviennent des transactions, et servent à ajuster l'état du portefeuille et du compte bancaire. Elles transitent aussi vers le service de notification pour informer l'utilisateur, et vers les composants d'affichage comme la **InfoBar** pour mise à jour en temps réel.

De plus, les indicateurs boursiers tels que les moyennes mobiles ou les bandes de Bollinger sont recalculés à partir des données de prix stockées dans les séries temporelles (XYSeries) du **GraphiquePanel**, assurant un affichage dynamique des variations de marché.

4.3 Dépendances d'éxecutions

L'interface graphique, notamment les classes **DetailsPanel** et **ListPanel**, dépendent des composants métiers tels que le Portefeuille, le **CompteBancaire**, ou encore le **Ser-**

viceNotification. Ces derniers sont sollicités pour exécuter les opérations courantes, comme acheter ou vendre.

Les modules métiers, de leur côté, dépendent de l'accès aux données de marché fournies par **DonnéesAlphaVantage**, qui constitue une dépendance externe liée à l'API.

Enfin, le mécanisme de notification et d'affichage repose sur une coordination entre le corps métier et l'interface, à travers l'appel à ServiceNotification et à la mise à jour de InfoBar.

5 Principaux choix de conception et réalisation

5.1 Importation et stockage des données

Pour développer cette application, il était nécessaire d'avoir accès à un grand volume de données de produits financiers, en particulier parce que l'utilisateur devait pouvoir accéder à n'importe quel produit de son choix. Pour cela, j'ai utilisé une API permettant de récupérer les données d'une action au format JSON à partir d'une clé et du symbole de l'action. Une méthode a été développée pour traiter ces données et les convertir en un tableau, ce qui permet de les exploiter efficacement. Ainsi, l'utilisateur peut simplement saisir le nom d'une action dans la barre de recherche pour observer son cours ou en acheter une quantité donnée.

Dans une première version, les données n'étaient pas directement stockées dans un fichier JSON une fois importées. Il fallait effectuer une requête vers l'API à chaque redémarrage de l'application ou à chaque affichage d'un nouveau graphique.

Cette approche présentait deux inconvénients :
- L'application était ralentie par les délais de réponse des requêtes API.
- Le quota de 20 requêtes API était rapidement atteint, empêchant toute action supplémentaire.

Pour résoudre ces problèmes, j'ai choisi de stocker localement les données d'une action dans un fichier JSON dès leur importation.

Ainsi, à chaque tentative d'importation d'une nouvelle action, l'application vérifie d'abord si un fichier JSON correspondant existe déjà, afin d'éviter une nouvelle requête API inutile.

5.2 Mise à jour dynamique des données et interaction utilisateur

Une fonctionnalité clé de cette application est sa capacité à réagir dynamiquement aux actions de l'utilisateur. Grâce à la `ToolBar`, l'utilisateur peut rechercher une action en saisissant son symbole, puis en appuyant sur "Entrée" ou sur le bouton `Rechercher`, ce qui déclenche la récupération ou le chargement local des données.

Chaque action recherchée est ajoutée à une liste interactive sur la gauche de l'écran. En cliquant sur une action, le graphique central est mis à jour automatiquement, tout

comme les informations affichées à droite (symbole, prix courant, prix sur le graphique, variation).

De plus, lors de l'achat d'un produit, l'utilisateur peut choisir entre le prix courant ou celui affiché sur le graphique à un instant précis (simulé via le bouton `Avancer`). Ce niveau de précision améliore le réalisme du simulateur, en permettant d'acheter une action à un moment donné du temps simulé.

6 Organisation du développement

Au début du développement, j'ai pris le temps de définir les objectifs du projet et de concevoir l'architecture UML. J'ai structuré le projet en deux grandes parties : l'interface graphique (IHM) et le cœur métier, comprenant la gestion des utilisateurs et des portefeuilles.

Pour organiser efficacement les tâches, j'ai utilisé l'outil Trello. Cela m'a permis de découper le développement en user-stories claires, facilitant le suivi de progression et la priorisation des fonctionnalités.

Même si certaines parties ont été développées à distance, j'ai constaté que travailler en présentiel, dans les salles de l'école, sur des postes côte à côte avec d'autres étudiants (sans lien direct avec ce projet), m'aidait à maintenir une bonne productivité. Ces sessions m'ont permis de me concentrer, de coder plus efficacement, et de tester les interactions entre les classes dans un environnement propice.

Cette organisation m'a permis de développer une application complète, avec une structure modulaire facilitant les extensions futures.